14

Authentication: Adding users to your application with Identity

This chapter covers

- How authentication works in web apps in ASP.NET Core
- Creating a project using the ASP.NET Core Identity system
- Adding user functionality to an existing web app
- Customizing the default ASP.NET Core Identity UI

One of the selling points of a web framework like ASP.NET Core is the ability to provide a dynamic app, customized to individual users. Many apps have the concept of an "account" with the service, which you can "sign in" to and get a different experience.

Depending on the service, an account gives you varying things: on some apps you may have to sign in to get access to additional features, and on others you might see suggested articles. On an e-commerce app, you'd be able to place orders and view your past orders; on Stack Overflow you can post questions and answers; whereas on a news site you might get a customized experience based on previous articles you've viewed.

When you think about adding users to your application, you typically have two aspects to consider:

- *Authentication*—The process of creating users and letting them log in to your app
- *Authorization*—Customizing the experience and controlling what users can do, based on the current logged-in user

In this chapter I'm going to be discussing the first of these points, authentication and membership, and in the next chapter I'll tackle the second point, authorization. In section 14.1 I discuss the difference between authentication and authorization, how authentication works in a traditional ASP.NET Core web app, and ways you can architect your system to provide sign-in functionality.

I also touch on the typical differences in authentication between a traditional web app and Web APIs used with client-side or mobile web apps. This book focuses on traditional web apps for authentication, but many of the principles are applicable to both.

In section 14.2 I introduce a user-management system called ASP.NET Core Identity (or Identity for short). Identity integrates with EF Core and provides services for creating and managing users, storing and validating passwords, and signing users in and out of your app.

In section 14.3 you'll create an app using a default template that includes ASP.NET Core Identity out of the box. This will give you an app to explore, to see the features Identity provides, as well as everything it doesn't.

Creating an app is great for seeing how the pieces fit together, but you'll often need to add users and authentication to an existing app. In section 14.4 you'll see the steps required to add ASP.NET Core Identity to an existing app: the recipe application from chapters 12 and 13.

In sections 14.5 and 14.6 you'll learn how to replace pages from the default Identity UI by "scaffolding" individual pages. In section 14.5 you'll see how to customize the Razor templates to generate different HTML on the user registration page, and in section 14.6 you'll learn how to customize the logic associated with a Razor Page. You'll see how to store additional information about a user (such as their name or date of birth) and how to provide them with permissions that you can later use to customize the app's behavior (if the user is a VIP, for example).

Before we look at the ASP.NET Core Identity system specifically, let's take a look at authentication and authorization in ASP.NET Core—what's happening when you sign in to a website and how you can design your apps to provide this functionality.

14.1 Introducing authentication and authorization

When you add sign-in functionality to your app and control access to certain functions based on the currently signed-in user, you're using two distinct aspects of security:

- *Authentication*—The process of determining *who you are*
- *Authorization*—The process of determining *what you're allowed to do*

Generally you need to know *who* the user is before you can determine *what* they're allowed to do, so authentication always comes first, followed by authorization. In this chapter, we're only looking at authentication; we'll cover authorization in chapter 15.

In this section I'll start by discussing how ASP.NET Core thinks about users and cover some of the terminology and concepts that are central to authentication. I always found this to be the hardest part to grasp when first learning about authentication, so I'll take it slow.

Next we'll look at what it means to sign in to a traditional web app. After all, you only provide your password and sign in to an app on a single page—how does the app know the request came from *you* for subsequent requests?

Finally, we'll look at how authentication works when you need to support client-side apps and mobile apps that call Web APIs, in addition to traditional web apps. Many of the concepts are similar, but the requirement to support multiple types of users, traditional apps, client-side apps, and mobile apps has led to alternative solutions.

14.1.1 Understanding users and claims in ASP.NET Core

The concept of a user is baked in to ASP.NET Core. In chapter 3, you learned that the HTTP server, Kestrel, creates an `HttpContext` object for every request it receives. This object is responsible for storing all the details related to that request, such as the request URL, any headers sent, the body of the request, and so on.

The `HttpContext` object also exposes the current *principal* for a request as the `User` property. This is ASP.NET Core's view of which user made the request. Any time your app needs to know who the current user is, or what they're allowed to do, it can look at the `HttpContext.User` principal.

DEFINITION You can think of the *principal* as the user of your app.

In ASP.NET Core, principals are implemented as `ClaimsPrincipals`, which has a collection of *claims* associated with it, as shown in figure 14.1.

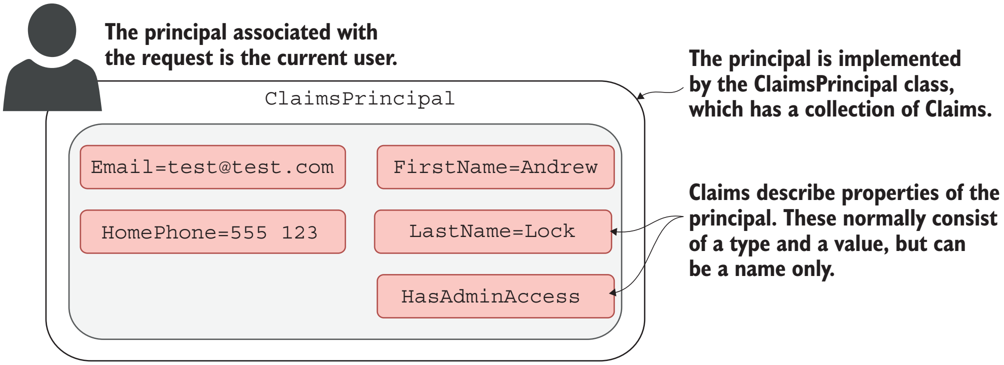

The principal associated with the request is the current user.
ClaimsPrincipal
Email=test@test.com
FirstName=Andrew
HomePhone=555 123
LastName=Lock
HasAdminAccess
The principal is implemented by the ClaimsPrincipal class, which has a collection of Claims.
Claims describe properties of the principal. These normally consist of a type and a value, but can be a name only.

Figure 14.1 The principal is the current user, implemented as `ClaimsPrincipal`. It contains a collection of `Claims` that describe the user.

You can think about claims as properties of the current user. For example, you could have claims for things like email, name, or date of birth.

> **DEFINITION** A claim is a single piece of information about a principal; it consists of a *claim type* and an optional *value*.

Claims can also be indirectly related to permissions and authorization, so you could have a claim called `HasAdminAccess` or `IsVipCustomer`. These would be stored in exactly the same way—as claims associated with the user principal.

> **NOTE** Earlier versions of ASP.NET used a role-based approach to security, rather than claims-based. The `ClaimsPrincipal` used in ASP.NET Core is compatible with this approach for legacy reasons, but you should use the claims-based approach for new apps.

Kestrel assigns a user principal to every request that arrives at your app. Initially that principal is a generic, anonymous, unauthenticated principal with no claims. How do you log in, and how does ASP.NET Core know that you've logged in on subsequent requests?

In the next section, we'll look at how authentication works in a traditional web app using ASP.NET Core, and the process of signing in to a user account.

14.1.2 Authentication in ASP.NET Core: Services and middleware

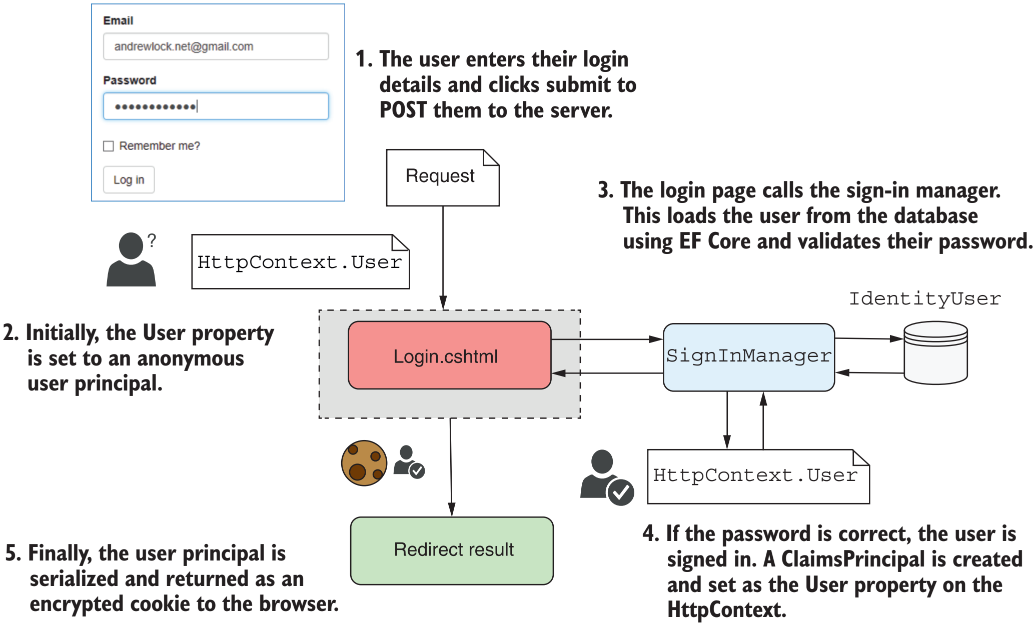
Adding authentication to any web app involves a number of moving parts. The same general process applies whether you're building a traditional web app or a client-side app, though there are often differences in implementation, as I'll discuss in section 14.1.3:

1. The client sends an identifier and a secret to the app, which identify the current user. For example, you could send an email address (identifier) and a password (secret).
2. The app verifies that the identifier corresponds to a user known by the app and that the corresponding secret is correct.
3. If the identifier and secret are valid, the app can set the principal for the current request, but it also needs a way of storing these details for subsequent requests. For traditional web apps, this is typically achieved by storing an encrypted version of the user principal in a cookie.

This is the typical flow for most web apps, but in this section I'm going to look at how it works in ASP.NET Core. The overall process is the same, but it's good to see how this pattern fits into the services, middleware, and MVC aspects of an ASP.NET Core application. We'll step through the various pieces at play in a typical app when you sign in as a user, what that means, and how you can make subsequent requests as that user.

SIGNING IN TO AN ASP.NET CORE APPLICATION

When you first arrive on a site and sign in to a traditional web app, the app will send you to a sign-in page and ask you to enter your username and password. After you submit the form to the server, the app redirects you to a new page, and you're magically logged in! Figure 14.2 shows what's happening behind the scenes in an ASP.NET Core app when you submit the form.



1. The user enters their login details and clicks submit to POST them to the server.

2. Initially, the User property is set to an anonymous user principal.

3. The login page calls the sign-in manager. This loads the user from the database using EF Core and validates their password.

4. If the password is correct, the user is signed in. A ClaimsPrincipal is created and set as the User property on the HttpContext.

5. Finally, the user principal is serialized and returned as an encrypted cookie to the browser.

Figure 14.2 Signing in to an ASP.NET Core application. `SignInManager` is responsible for setting `HttpContext.User` to the new principal and serializing the principal to the encrypted cookie.

This shows the series of steps from the moment you submit the login form on a Razor Page to the point the redirect is returned to the browser. When the request first arrives, Kestrel creates an anonymous user principal and assigns it to the `HttpContext.User` property. The request is then routed to the Login.cshtml Razor Page, which reads the email and password from the request using model binding.

The meaty work happens inside the `SignInManager` service. This is responsible for loading a user entity with the provided username from the database and validating that the password they provided is correct.

> **WARNING** Never store passwords in the database directly. They should be *hashed* using a strong one-way algorithm. The ASP.NET Core Identity system does this for you, but it's always wise to reiterate this point!

If the password is correct, SignInManager creates a new ClaimsPrincipal from the user entity it loaded from the database and adds the appropriate claims, such as the email address. It then replaces the old, anonymous HttpContext.User principal with the new, authenticated principal.

Finally, SignInManager serializes the principal, encrypts it, and stores it as a *cookie*. A cookie is a small piece of text that's sent back and forth between the browser and your app along with each request, consisting of a name and a value.

This authentication process explains how you can set the user for a request when they *first* log in to your app, but what about subsequent requests? You only send your password when you first log in to an app, so how does the app know that it's the same user making the request?

Authenticating users for subsequent requests

The key to persisting your identity across multiple requests lies in the final step of figure 14.2, where you serialized the principal in a cookie. Browsers will automatically send this cookie with all requests made to your app, so you don't need to provide your password with every request.

ASP.NET Core uses the authentication cookie sent with the requests to rehydrate ClaimsPrincipal and set the HttpContext.User principal for the request, as shown in figure 14.3. The important thing to note is *when* this process happens—in the AuthenticationMiddleware.

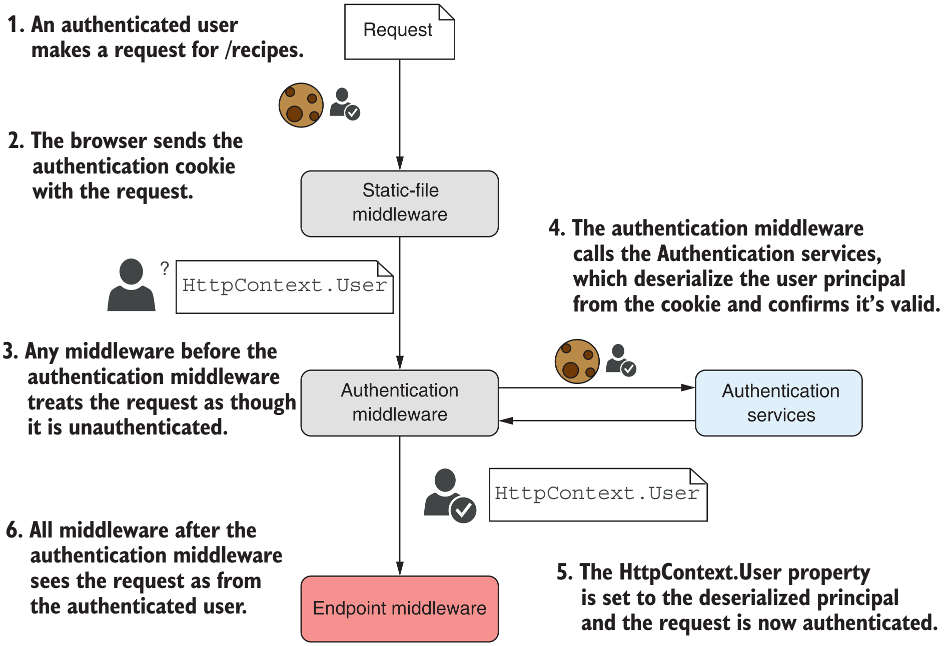

1. An authenticated user makes a request for /recipes.
Request
2. The browser sends the authentication cookie with the request.
Static-file middleware
?
HttpContext.User
3. Any middleware before the authentication middleware treats the request as though it is unauthenticated.
4. The authentication middleware calls the Authentication services, which deserialize the user principal from the cookie and confirms it's valid.
Authentication middleware
Authentication services
HttpContext.User
5. The HttpContext.User property is set to the deserialized principal and the request is now authenticated.
6. All middleware after the authentication middleware sees the request as from the authenticated user.
Endpoint middleware

Figure 14.3 A subsequent request after signing in to an application. The cookie sent with the request contains the user principal, which is validated and used to authenticate the request.

When a request containing the authentication cookie is received, Kestrel creates the default, unauthenticated, anonymous principal and assigns it to the `HttpContext.User` principal. Any middleware that runs at this point, before `AuthenticationMiddleware`, will see the request as unauthenticated, even if there's a valid cookie.

> **TIP** If it looks like your authentication system isn't working, double-check your middleware pipeline. Only middleware that runs after `AuthenticationMiddleware` will see the request as authenticated.

`AuthenticationMiddleware` is responsible for setting the current user for a request. The middleware calls the authentication services, which reads the cookie from the request, decrypts it, and deserializes it to obtain the `ClaimsPrincipal` created when the user logged in.

`AuthenticationMiddleware` sets the `HttpContext.User` principal to the new, authenticated principal. All subsequent middleware will now know the user principal for the request and can adjust their behavior accordingly (for example, displaying the user's name on the home page, or restricting access to some areas of the app).

> **NOTE** The `AuthenticationMiddleware` is *only* responsible for authenticating incoming requests and setting the `ClaimsPrincipal` if the request contains an authentication cookie. It is *not* responsible for redirecting unauthenticated requests to the login page or rejecting unauthorized requests—that is handled by the `AuthorizationMiddleware`, as you'll see in chapter 15.

The process described so far, in which a single app authenticates the user when they log in and sets a cookie that's read on subsequent requests, is common with traditional web apps, but it isn't the only possibility. In the next section we'll take a look at authentication for Web API applications, used by client-side and mobile apps, and at how the authentication system changes for those scenarios.

14.1.3 Authentication for APIs and distributed applications

The process I've outlined so far applies to traditional web apps, where you have a single endpoint that's doing all the work. It's responsible for authenticating and managing users, as well as serving your app data, as shown in figure 14.4.

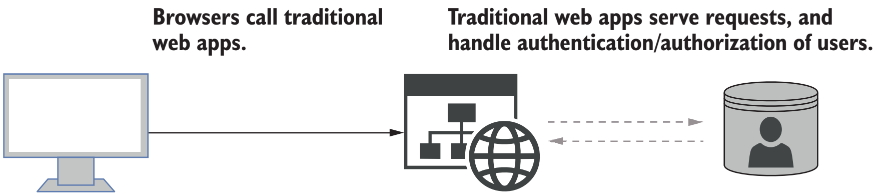

Browsers call traditional web apps.
Traditional web apps serve requests, and handle authentication/authorization of users.

Figure 14.4 Traditional apps typically handle all the functionality of an app: the business logic, generating the UI, authentication, and user management.

In addition to this traditional web app, it's common to use ASP.NET Core as a Web API to serve data for mobile and client-side SPAs. Similarly, the trend toward microservices on the backend means that even traditional web apps using Razor often need to call APIs behind the scenes, as shown in figure 14.5.

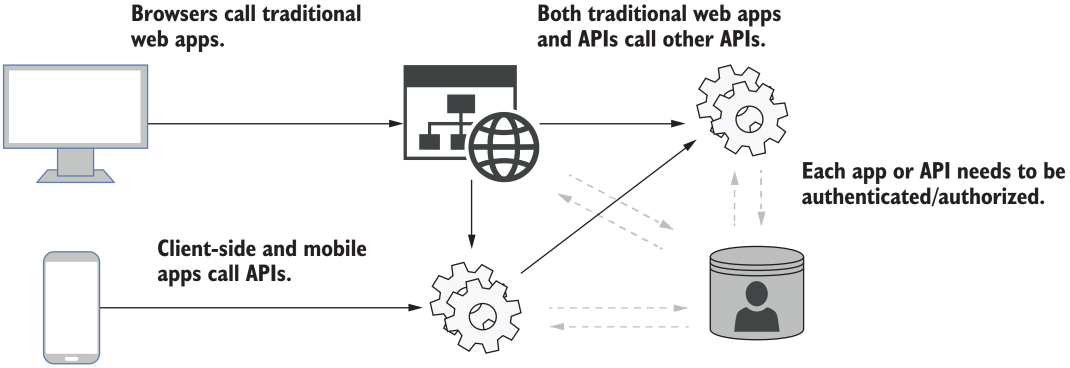

Browsers call traditional web apps.
Both traditional web apps and APIs call other APIs.
Each app or API needs to be authenticated/authorized.
Client-side and mobile apps call APIs.

Figure 14.5 Modern applications typically need to expose Web APIs for mobile and client-side apps, as well as potentially calling APIs on the backend. When all of these services need to authenticate and manage users, this becomes logistically complicated.

In this situation, you have multiple apps and APIs, all of which need to understand that the same user is making a request across all the apps and APIs. If you keep the same approach as before, where each app manages its own users, things can quickly become unmanageable!

You'd need to duplicate all the sign-in logic between the apps and APIs, as well as needing to have some central database holding the user details. Users may need to sign in multiple times to access different parts of the service. On top of that, using cookies becomes problematic for some mobile clients in particular, or where you're making requests to multiple domains (as cookies only belong to a single domain).

How can you improve this? The typical approach is to extract the code that's common to all of the apps and APIs, and move it to an *identity provider*, as shown in figure 14.6.

Instead of signing in to an app directly, the app redirects to an identity provider app. The user signs in to this identity provider, which passes *bearer tokens* back to the client that indicate who the user is and what they're allowed to access. The clients and apps can pass these tokens to the APIs to provide information about the logged-in user, without needing to re-authenticate or manage users directly.

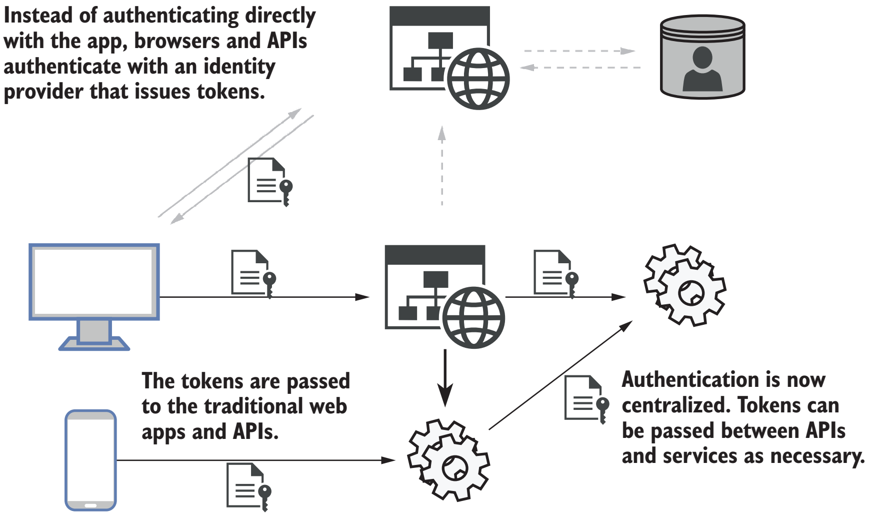

Instead of authenticating directly with the app, browsers and APIs authenticate with an identity provider that issues tokens.
The tokens are passed to the traditional web apps and APIs.
Authentication is now centralized. Tokens can be passed between APIs and services as necessary.

Figure 14.6 An alternative architecture involves using a central identity provider to handle all the authentication and user management for the system. Tokens are passed back and forth between the identity provider, apps, and APIs.

This architecture is clearly more complicated on the face of it, as you've thrown a whole new service—the identity provider—into the mix, but in the long run this has a number of advantages:

- *Users can share their identity between multiple services.* As you're logged in to the central identity provider, you're essentially logged in to *all* apps that use that service. This gives you the single-sign-on experience, where you don't have to keep logging in to multiple services.
- *Reduced duplication.* All of the sign-in logic is encapsulated in the identity provider, so you don't need to add sign-in screens to all your apps.
- *Can easily add new providers.* Whether you use the identity provider approach or the traditional approach, it's possible to use external services to handle the authentication of users. You'll have seen this on apps that allow you to "log in using Facebook" or "log in using Google," for example. If you use a centralized identity provider, adding support for additional providers can be handled in one place, instead of having to configure every app and API explicitly.

Out of the box, ASP.NET Core supports architectures like this, and for *consuming* issued bearer tokens, but it doesn't include support for *issuing* those tokens in the core framework. That means you'll need to use another library or service for the identity provider.

One option for an identity provider is to delegate all the authentication responsibilities to a third-party identity provider, such as Facebook, Okta, Auth0, or Azure Active

Directory B2C. These manage users for you, so user information and passwords are stored in their database, rather than your own. The biggest advantage of this approach is that you don't have to worry about making sure your customer data is safe; you can be pretty sure that a third party will protect it, as it's their whole business.

> **TIP** Wherever possible, I recommend this approach, as it delegates security responsibilities to someone else. You can't lose your user's details if you never had them!

Another common option is to build your own identity provider. This may sound like a lot of work, but thanks to excellent libraries like OpenIddict (https://github.com/openiddict) and IdentityServer (https://docs.identityserver.io/), it's perfectly possible to write your own identity provider to serve bearer tokens that will be consumed by an application.

An aspect often overlooked by people getting started with OpenIddict and IdentityServer is that they aren't prefabricated solutions. You, as a developer, need to write the code that knows how to create a new user (normally in a database), how to load a user's details, and how to validate their password. In that respect, the development process of creating an identity provider is similar to the traditional web app with cookie authentication that I discussed in section 14.1.2.

In fact, you can almost think of an identity provider as a traditional web app that only has account management pages. It also has the ability to generate tokens for other services, but it contains no other app-specific logic. The need to manage users in a database, as well as provide an interface for users to log in, is common to both approaches and is the focus of this chapter.

> **NOTE** Hooking up your apps and APIs to use an identity provider can require a fair amount of tedious configuration, both of the app and the identity provider. For simplicity, this book focuses on traditional web apps using the process outlined in section 14.1.2. ASP.NET Core includes a helper library for working with IdentityServer and client-side SPAs. For details on how to get started, see Microsoft's "Authentication and authorization for SPAs" documentation at http://mng.bz/w9Mq and the IdentityServer docs: https://docs.identityserver.io/.

ASP.NET Core Identity (hereafter shortened to Identity) is a system that makes building the user-management aspect of your app (or identity provider app) simpler. It handles all of the boilerplate for saving and loading users to a database, as well as a number of best practices for security, such as user lock-out, password hashing, and *two-factor authentication.*

> **DEFINITION** *Two-factor authentication* (2FA) is where you require an extra piece of information to sign in, in addition to a password. This could involve sending a code to a user's phone by SMS, or using a mobile app to generate a code, for example.

In the next section I'm going to talk about the ASP.NET Core Identity system, the problems it solves, when you'd want to use it, and when you might not want to use it. In section 14.3 we'll take a look at some code and see ASP.NET Core Identity in action.

14.2 What is ASP.NET Core Identity?

Whether you're writing a traditional web app using Razor Pages or are setting up a new identity provider using a library like IdentityServer, you'll need a way of persisting details about your users, such as their usernames and passwords.

This might seem like a relatively simple requirement, but, given that this is related to security and people's personal details, it's important you get it right. As well as storing the claims for each user, it's important to store passwords using a strong hashing algorithm to allow users to use 2FA where possible, and to protect against brute force attacks, to name a few of the many requirements. Although it's perfectly possible to write all the code to do this manually and to build your own authentication and membership system, I highly recommend you don't.

I've already mentioned third-party identity providers such as Auth0 or Azure Active Directory B2C. These are Software-as-a-Service (SaaS) solutions that take care of the user-management and authentication aspects of your app for you. If you're in the process of moving apps to the cloud generally, then solutions like these can make a lot of sense.

If you can't or don't want to use these third-party solutions, I recommend you consider using the ASP.NET Core Identity system to store and manage user details in your database. ASP.NET Core Identity takes care of most of the boilerplate associated with authentication, but it remains flexible and lets you control the login process for users if you need to.

NOTE ASP.NET *Core* Identity is an evolution of ASP.NET Identity, with some design improvements and converted to work with ASP.NET Core.

By default, ASP.NET Core Identity uses EF Core to store user details in the database. If you're already using EF Core in your project, this is a perfect fit. Alternatively, it's possible to write your own stores for loading and saving user details in another way.

Identity takes care of the low-level parts of user management, as shown in table 14.1. As you can see from this list, Identity gives you a lot, but not everything—by a long shot!

Table 14.1 Which services are and aren't handled by ASP.NET Core Identity

Managed by ASP.NET Core Identity	Requires implementing by the developer
Database schema for storing users and claims.	UI for logging in, creating, and managing users (Razor Pages or controllers). This is included in an optional package, which provides a default UI.
Creating a user in the database.	Sending email messages.
Password validation and rules.	Customizing claims for users (adding new claims).

Table 14.1 Which services are and aren't handled by ASP.NET Core Identity *(continued)*

Managed by ASP.NET Core Identity	Requires implementing by the developer
Handling user account lockout (to prevent brute-force attacks).	Configuring third-party identity providers.
Managing and generating 2FA codes.	
Generating password-reset tokens.	
Saving additional claims to the database.	
Managing third-party identity providers (for example, Facebook, Google, Twitter).	

The biggest missing piece is the fact that you need to provide all the UI for the application, as well as tying all the individual Identity services together to create a functioning sign-in process. That's a pretty big missing piece, but it makes the Identity system extremely flexible.

Luckily, ASP.NET Core includes a helper NuGet library, Microsoft.AspNetCore.Identity.UI, that gives you the whole of the UI boilerplate for free. That's over 30 Razor Pages with functionality for logging in, registering users, using two-factor authentication, and using external login providers, among others. You can still customize these pages if you need to, but having a whole login process working out of the box, with no code required on your part, is a huge win. We'll look at this library and how you use it in sections 14.3 and 14.4.

For that reason, I strongly recommend using the default UI as a starting point, whether you're creating an app or adding user management to an existing app. But the question still remains: when should you use Identity, and when should you consider rolling your own?

I'm a big fan of Identity, so I tend to suggest it in most situations, as it handles a lot of security-related things for you that are easy to mess up. I've heard several arguments against it, some of which are valid, and others less so:

- *I already have user authentication in my app*—Great! In that case, you're probably right, Identity may not be necessary. But does your custom implementation use 2FA? Do you have account lockout? If not, and you need to add them, then considering Identity may be worthwhile.
- *I don't want to use EF Core*—That's a reasonable stance. You could be using Dapper, some other ORM, or even a document database for your database access. Luckily, the database integration in Identity is pluggable, so you could swap out the EF Core integration and use your own database integration libraries instead.
- *My use case is too complex for Identity*—Identity provides lower-level services for authentication, so you can compose the pieces however you like. It's also extensible, so if you need to, for example, transform claims before creating a principal, you can.

- *I don't like the default Razor Pages UI*—The default UI for Identity is entirely optional. You can still use the Identity services and user management but provide your own UI for logging in and registering users. However, be aware that although doing this gives you a lot of flexibility, it's also very easy to introduce a security flaw in your user-management system—the last place you want security flaws!
- *I'm not using Bootstrap to style my application*—The default Identity UI uses Bootstrap as a styling framework, the same as the default ASP.NET Core templates. Unfortunately, you can't easily change that, so if you're using a different framework, or you need to customize the HTML generated, you can still use Identity but you'll need to provide your own UI.
- *I don't want to build my own identity system*—I'm glad to hear it. Using an external identity provider like Azure Active Directory B2C or Auth0 is a great way of shifting the responsibility and risk associated with storing users' personal information onto a third party.

Any time you're considering adding user management to your ASP.NET Core application, I'd recommend looking at Identity as a great option for doing so. In the next section I'll demonstrate what Identity provides by creating a new Razor Pages application using the default Identity UI. In section 14.4 we'll take that template and apply it to an existing app instead, and in sections 14.5 and 14.6 you'll see how to override the default pages.

14.3 Creating a project that uses ASP.NET Core Identity

I've covered authentication and Identity in general terms, but the best way to get a feel for it is to see some working code. In this section we're going to look at the default code generated by the ASP.NET Core templates with Identity, how the project works, and where Identity fits in.

14.3.1 Creating the project from a template

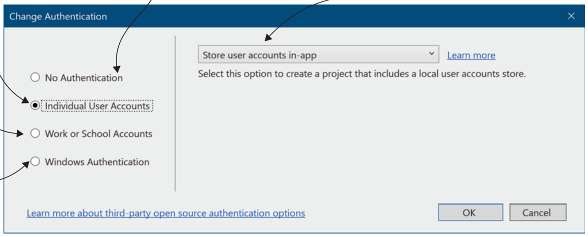
You'll start by using the Visual Studio templates to generate a simple Razor Pages application that uses Identity for storing individual user accounts in a database.

> **TIP** You can create an equivalent project using the .NET CLI by running `dotnet new webapp -au Individual -uld`.

To create the template using Visual Studio, you must be using VS 2019 or later and have the .NET 5.0 SDK installed:

1. Choose File > New > Project or choose Create a New Project from the splash screen.
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template.
3. On the next screen, enter a project name, location, and a solution name, and click Create.

4. Choose the Web Application template and click Change under Authentication to bring up the Authentication dialog box, shown in figure 14.7.



Change Authentication

○ No Authentication
● Individual User Accounts
○ Work or School Accounts
○ Windows Authentication

Store user accounts in-app — Learn more

Select this option to create a project that includes a local user accounts store.

Learn more about third-party open source authentication options

Choose Individual User Accounts to store local user accounts using ASP.NET Core Identity and EF Core.

Choose No Authentication to create a template without authentication.

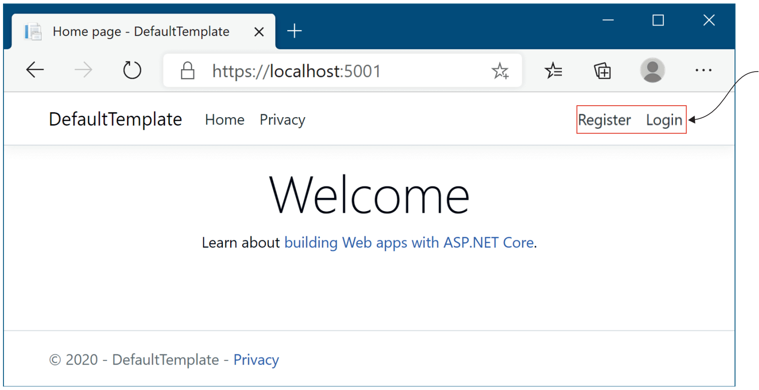
Choose Store User Accounts In-app.

Work or School Accounts will configure the application to use an external Identity Provider—using Active Directory (or Office 365, for example) to handle user management and authentication.

Choose Windows Authentication for intranet sites where the Windows login of the user provides the authentication mechanism.

Figure 14.7 Choosing the authentication mode of the new ASP.NET Core application template in VS 2019

5. Choose Individual User Accounts to create an application configured with EF Core and ASP.NET Core Identity. Click OK.
6. Click Create to create the application. Visual Studio will automatically run `dotnet restore` to restore all the necessary NuGet packages for the project.
7. Run the application to see the default app, as shown in figure 14.8.



Welcome

Learn about building Web apps with ASP.NET Core.

© 2020 - DefaultTemplate - Privacy

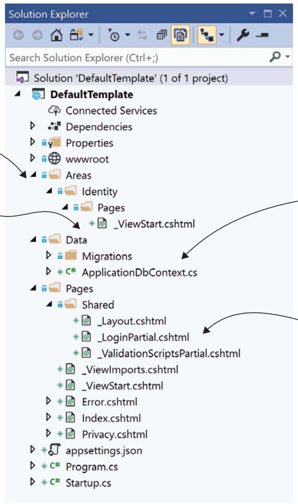
You can create new users and sign in using the Login widget.

Figure 14.8 The default template with individual account authentication looks similar to the no authentication template, with the addition of a Login widget at the top right of the page.

This template should look familiar, with one twist: you now have Register and Login buttons! Feel free to play with the template—creating a user, logging in and out—to get a feel for the app. Once you're happy, look at the code generated by the template and the boilerplate it saved you from writing.

14.3.2 Exploring the template in Solution Explorer

The project generated by the template, shown in figure 14.9, is very similar to the default no-authentication template. That's largely due to the default UI library, which brings in a big chunk of functionality without exposing you to the nitty-gritty details.



The ASP.NET Core Identity default UI is contained in an Area called Identity.

You can override individual files in the default UI by placing files in corresponding locations in the Areas/Identity folder.

The template includes an EF Core DbContext and migrations to configure the database schema for ASP.NET Core Identity.

The Pages folder is identical to the no-authentication template, with an additional _LoginPartial view.

Figure 14.9 The project layout of the default template. Depending on your version of Visual Studio, the exact files may vary slightly.

The biggest addition is the Areas folder in the root of your project, which contains an Identity subfolder. Areas are sometimes used for organizing sections of functionality. Each area can contain its own Pages folder, which is analogous to the main Pages folder in your application.

DEFINITION *Areas* are used to group Razor Pages into separate hierarchies for organizational purposes. I rarely use areas and prefer to create subfolders in the main Pages folder instead. The one exception is the Identity UI, which uses a separate Identity area by default. For more details on areas, see Microsoft's "Areas in ASP.NET Core" documentation: http://mng.bz/7Vw9.

The Microsoft.AspNetCore.Identity.UI package creates Razor Pages in the Identity area. You can override any page in this default UI by creating a corresponding page in the

Areas/Identity/Pages folder in your application. For example, as shown in figure 14.9, the default template adds a _ViewStart.cshtml file that overrides the version that is included as part of the default UI. This file contains the following code, which sets the default Identity UI Razor Pages to use your project's default _Layout.cshtml file:

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
}
```

Some obvious questions at this point might be "how do you know what's included in the default UI," and "which files you can override"? You'll see the answer to both of those in section 14.5, but in general you should try to avoid overriding files where possible. After all, the goal with the default UI is to *reduce* the amount of code you have to write!

The Data folder in your new project template contains your application's EF Core `DbContext`, called `ApplicationDbContext`, and the migrations for configuring the database schema to use Identity. I'll discuss this schema in more detail in section 14.3.3.

The final additional file included in this template compared to the no-authentication version is the partial Razor view Pages/Shared/_LoginPartial.cshtml. This provides the Register and Login links you saw in figure 14.8, and it's rendered in the default Razor layout, _Layout.cshtml.

If you look inside _LoginPartial.cshtml, you can see how routing works with areas by combining the Razor Page path with an `{area}` route parameter using Tag Helpers. For example, the Login link specifies that the Razor Page `/Account/Login` is in the `Identity` area using the `asp-area` attribute:

```
<a asp-area="Identity" asp-page="/Account/Login">Login</a>
```

TIP You can reference Razor Pages in the `Identity` area by setting the `area` route value to `Identity`. You can use the `asp-area` attribute in Tag Helpers that generate links.

In addition to the new files included thanks to ASP.NET Core Identity, it's worth opening up Startup.cs and looking at the changes there. The most obvious change is the additional configuration in `ConfigureServices`, which adds all the services Identity requires.

Listing 14.1 Adding ASP.NET Core Identity services to `ConfigureServices`

Adds the Identity system, including the default UI, and configures the user type as IdentityUser

ASP.NET Core Identity uses EF Core, so it includes the standard EF Core configuration.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
            options.SignIn.RequireConfirmedAccount = true)
          .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}
```

Requires users to confirm their accounts (typically by email) before they log in

Configures Identity to store its data in EF Core

The `AddDefaultIdentity()` extension method does several things:

- Adds the core ASP.NET Core Identity services.
- Configures the application user type to be `IdentityUser`. This is the entity model that is stored in the database and represents a "user" in your application. You can extend this type if you need to, but that's not always necessary, as you'll see in section 14.6.
- Adds the default UI Razor Pages for registering, logging in, and managing users.
- Configures token providers for generating 2FA and email confirmation tokens.

There's another, very important change in `Startup`, in the `Configure` method:

```
app.UseAuthentication();
```

This adds `AuthenticationMiddleware` to the pipeline, so that you can authenticate incoming requests, as you saw in figure 14.3. *The location of this middleware is very important.* It should be placed after `UseRouting()` and before `UseAuthorization()` and `UseEndpoints()`, as shown in the following listing.

Listing 14.2 Adding `AuthenticationMiddleware` to your middleware pipeline

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Middleware placed before UseAuthentication will see all requests as anonymous.

The routing middleware determines which page is requested based on the request URL.

UseAuthentication should be placed after UseRouting.

UseEndpoints should be last, after the user principal is set and authorization has been applied.

UseAuthorization should be placed after UseAuthentication so it can access the user principal.

If you don't use this specific order of middleware, you can run into strange bugs where users aren't authenticated correctly, or where authorization policies aren't correctly applied. This order is configured for you automatically in your templates, but it's something to be careful about if you're upgrading an existing application or moving middleware around.

> **IMPORTANT** `UseAuthentication()` and `UseAuthorization()` must be placed between `UseRouting()` and `UseEndpoints()`. Additionally, `UseAuthorization()` must be placed after `UseAuthentication()`. You can add additional middleware between each of these calls, as long as this overall middleware order is preserved.

Now that you've got an overview of the additions made by Identity, we'll look in a bit more detail at the database schema and how Identity stores users in the database.

14.3.3 The ASP.NET Core Identity data model

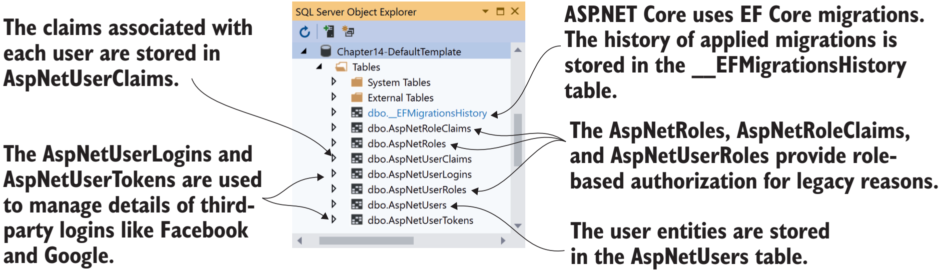
Out of the box, and in the default templates, Identity uses EF Core to store user accounts. It provides a base `DbContext` that you can inherit from, called `IdentityDbContext`, which uses an `IdentityUser` as the user entity for your application.

In the template, the app's `DbContext` is called `ApplicationDbContext`. If you open up this file, you'll see it's very sparse; it inherits from the `IdentityDbContext` base class I described earlier, and that's it. What does this base class give you? The easiest way to see is to update a database with the migrations and take a look.

Applying the migrations is the same process as in chapter 12. Ensure the connection string points to where you want to create the database, open a command prompt in your project folder, and run this command to update the database with the migrations:

```
dotnet ef database update
```

If the database doesn't yet exist, the CLI will create it. Figure 14.10 shows what the database looks like for the default template.[1]



The claims associated with each user are stored in AspNetUserClaims.

The AspNetUserLogins and AspNetUserTokens are used to manage details of third-party logins like Facebook and Google.

ASP.NET Core uses EF Core migrations. The history of applied migrations is stored in the __EFMigrationsHistory table.

The AspNetRoles, AspNetRoleClaims, and AspNetUserRoles provide role-based authorization for legacy reasons.

The user entities are stored in the AspNetUsers table.

Figure 14.10 The database schema used by ASP.NET Core Identity

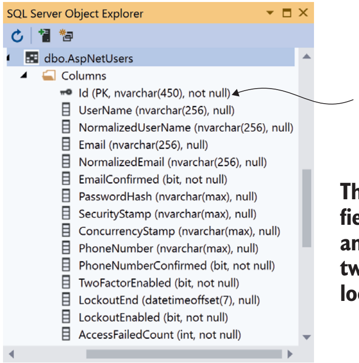
[1] If you're using MS SQL Server (or LocalDB), you can use the SQL Server Object Explorer in Visual Studio to browse tables and objects in your database. See Microsoft's "How to: Connect to a Database and Browse Existing Objects" for details: http://mng.bz/mg8r.

> **TIP** If you see an error after running the `dotnet ef` command, ensure you have the .NET tool installed by following the instructions provided in section 12.3.1. Also ensure you run the command from the *project* folder, not the *solution* folder.

That's a lot of tables! You shouldn't need to interact with these tables directly—Identity handles that for you—but it doesn't hurt to have a basic grasp of what they're for:

- *__EFMigrationsHistory*—The standard EF Core migrations table that records which migrations have been applied.
- *AspNetUsers*—The user profile table itself. This is where `IdentityUser` is serialized to. We'll take a closer look at this table shortly.
- *AspNetUserClaims*—The claims associated with a given user. A user can have many claims, so it's modeled as a many-to-one relationship.
- *AspNetUserLogins and AspNetUserTokens*—These are related to third-party logins. When configured, these let users sign in with a Google or Facebook account (for example), instead of creating a password on your app.
- *AspNetUserRoles, AspNetRoles, and AspNetRoleClaims*—These tables are somewhat of a legacy left over from the old role-based permission model of the pre-.NET 4.5 days, instead of the claims-based permission model. These tables let you define roles that multiple users can belong to. Each role can be assigned a number of claims. These claims are effectively inherited by a user principal when they are assigned that role.

You can explore these tables yourself, but the most interesting of them is the AspNetUsers table, shown in figure 14.11.



```
dbo.AspNetUsers
  Columns
    Id (PK, nvarchar(450), not null)
    UserName (nvarchar(256), null)
    NormalizedUserName (nvarchar(256), null)
    Email (nvarchar(256), null)
    NormalizedEmail (nvarchar(256), null)
    EmailConfirmed (bit, not null)
    PasswordHash (nvarchar(max), null)
    SecurityStamp (nvarchar(max), null)
    ConcurrencyStamp (nvarchar(max), null)
    PhoneNumber (nvarchar(max), null)
    PhoneNumberConfirmed (bit, not null)
    TwoFactorEnabled (bit, not null)
    LockoutEnd (datetimeoffset(7), null)
    LockoutEnabled (bit, not null)
    AccessFailedCount (int, not null)
```

By default, ASP.NET Core Identity uses Guids for the user Id, stored as strings in the database.

The table contains all the neccessary fields for authenticating a user, email and phone number confirmation, two-factor authentication, and account lockout.

Figure 14.11 The AspNetUsers table is used to store all the details required to authenticate a user.

Most of the columns in the AspNetUsers table are security related—the user's email, password hash, whether they have confirmed their email, whether they have 2FA enabled, and so on. By default, there are no columns for additional information, like the user's name.

> **NOTE** You can see from figure 14.11 that the primary key `Id` is stored as a `string` column. By default, Identity uses `Guid` for the identifier. To customize the data type, see the "Change the primary key type" section of Microsoft's "Identity model customization in ASP.NET Core" documentation: http://mng.bz/5jdB.

Any additional properties of the user are stored as claims in the AspNetUserClaims table associated with that user. This lets you add arbitrary additional information, without having to change the database schema to accommodate it. Want to store the user's date of birth? You could add a claim to that user—no need to change the database schema. You'll see this in action in section 14.6, when you add a Name claim to every new user.

> **NOTE** Adding claims is often the easiest way to extend the default `IdentityUser`, but you can also add additional properties to the `IdentityUser` directly. This requires database changes but is nevertheless useful in many situations. You can read how to add custom data using this approach here: http://mng.bz/Xd61.

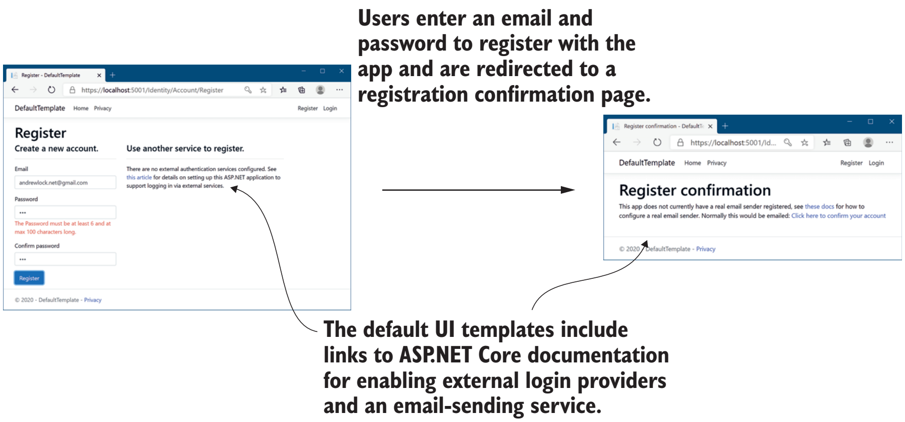
It's important to understand the difference between the `IdentityUser` entity (stored in the AspNetUsers table) and the `ClaimsPrincipal`, which is exposed on `HttpContext.User`. When a user first logs in, an `IdentityUser` is loaded from the database. This entity is combined with additional claims for the user from the AspNetUserClaims table to create a `ClaimsPrincipal`. It's this `ClaimsPrincipal` that is used for authentication and is serialized to the authentication cookie, not the `IdentityUser`.

It's useful to have a mental model of the underlying database schema Identity uses, but in day-to-day work, you shouldn't have to interact with it directly—that's what Identity is for, after all! In the next section, we'll look at the other end of the scale—the UI of the app, and what you get out of the box with the default UI.

14.3.4 Interacting with ASP.NET Core Identity

You'll want to explore the default UI yourself, to get a feel for how the pieces fit together, but in this section I'll highlight what you get out of the box, as well as areas that typically require additional attention right away.

The entry point to the default UI is the user registration page of the application, shown in figure 14.12. The register page enables users to sign up to your application by creating a new `IdentityUser` with an email and a password. After creating an account, users are redirected to a screen indicating that they should confirm their email. No email service is enabled by default, as this is dependent of you configuring an external email service. You can read how to enable email sending in Microsoft's "Account confirmation and password recovery in ASP.NET Core" documentation at



Register - DefaultTemplate

https://localhost:5001/Identity/Account/Register

DefaultTemplate Home Privacy Register Login

Register

Create a new account.

Email

andrewlock.net@gmail.com

Password

•••

The Password must be at least 6 and at max 100 characters long.

Confirm password

•••

Register

Use another service to register.

There are no external authentication services configured. See this article for details on setting up this ASP.NET application to support logging in via external services.

© 2020 - DefaultTemplate - Privacy

Register confirmation - DefaultTe

https://localhost:5001/Id...

DefaultTemplate Home Privacy Register Login

Register confirmation

This app does not currently have a real email sender registered, see these docs for how to configure a real email sender. Normally this would be emailed: Click here to confirm your account

© 2020 - DefaultTemplate - Privacy

Users enter an email and password to register with the app and are redirected to a registration confirmation page.

The default UI templates include links to ASP.NET Core documentation for enabling external login providers and an email-sending service.

Figure 14.12 The registration flow for users using the default Identity UI. Users enter an email and password and are redirected to a "confirm your email" page. This is a placeholder page by default, but if you enable email confirmation, this page will update appropriately.

http://mng.bz/6gBo. Once you configure this, users will automatically receive an email with a link to confirm their account.

By default, user emails must be unique (you can't have two users with the same email), and the password must meet various length and complexity requirements. You can customize these options and more in the configuration lambda of the call to `AddDefaultIdentity()` in Startup.cs, as shown in the following listing.

Listing 14.3 Customizing Identity settings in `ConfigureServices` in Startup.cs

```
services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = true;
    options.Lockout.AllowedForNewUsers = true;
    options.Password.RequiredLength = 12;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = false;
})
.AddEntityFrameworkStores<AppDbContext>();
```

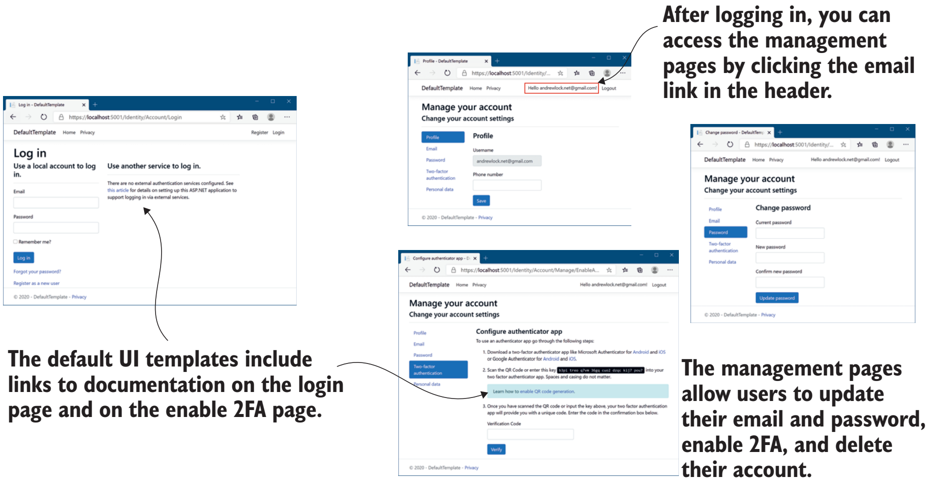
Require users to confirm their account by email before they can log in.

Update password requirements. Current guidance is to require long passwords.

Enables user-lockout, to prevent brute-force attacks against user passwords

After a user has registered with your application, they need to log in, as shown in figure 14.13. On the right side of the login page, the default UI templates describe how you, the developer, can configure external login providers, such as Facebook and Google. This is useful information for you, but it's one of the reasons you may need to customize the default UI templates, as you'll see in section 14.5.

Once a user has signed in, they can access the management pages of the identity UI. These allow users to change their email, change their password, configure 2FA



Log in

Use a local account to log in.

Use another service to log in.

There are no external authentication services configured. See this article for details on setting up this ASP.NET application to support logging in via external services.

Manage your account

Change your account settings

Configure authenticator app

To use an authenticator app go through the following steps:

1. Download a two-factor authenticator app like Microsoft Authenticator for Android and iOS or Google Authenticator for Android and iOS.
2. Scan the QR Code or enter this key b3pi treo q7vm 36gq cuo2 dzqc kij7 pou7 into your two factor authenticator app. Spaces and casing do not matter.

Learn how to enable QR code generation.

3. Once you have scanned the QR code or input the key above, your two factor authentication app will provide you with a unique code. Enter the code in the confirmation box below.

After logging in, you can access the management pages by clicking the email link in the header.

The default UI templates include links to documentation on the login page and on the enable 2FA page.

The management pages allow users to update their email and password, enable 2FA, and delete their account.

Figure 14.13 Logging in with an existing user and managing the user account. The Login page describes how to configure external login providers, such as Facebook and Google. The user-management pages allow users to change their email and password and to configure two-factor authentication (2FA).

with an authenticator app, or delete all their personal data. Most of these functions work without any effort on your part, assuming you've already configured an email-sending service.[2]

That covers everything you get in the default UI templates. It may seem somewhat minimal, but it covers a lot of the requirements that are common to almost all apps. Nevertheless, there are a few things you'll nearly always want to customize:

- Configure an email-sending service, to enable account confirmation and password recovery, as described in Microsoft's "Account confirmation and password recovery in ASP.NET Core" documentation: http://mng.bz/vzy7.
- Add a QR code generator for the enable 2FA page, as described in Microsoft's "Enable QR Code generation for TOTP authenticator apps in ASP.NET Core" documentation: http://mng.bz/4Zmw.
- Customize the register and login pages to remove the documentation link for enabling external services. You'll see how to do this in section 14.5. Alternatively, you may want to disable user registration entirely, as described in Microsoft's "Scaffold Identity in ASP.NET Core projects" documentation: http://mng.bz/QmMG.

[2] You can improve the 2FA authenticator page by enabling QR code generation, as described in Microsoft's "Enable QR Code generation for TOTP authenticator apps in ASP.NET Core" document: http://mng.bz/nM5a.

- Collect additional information about users on the registration page. You'll see how to do this in section 14.6.

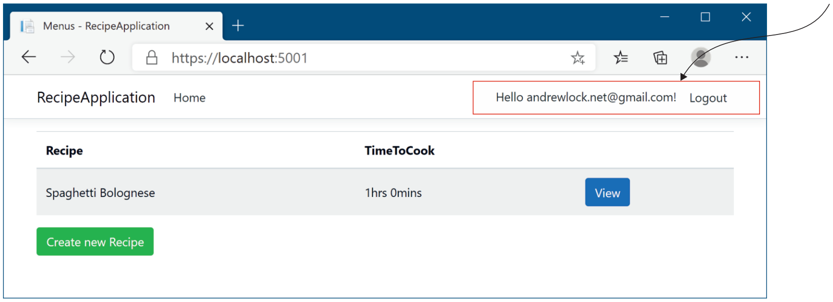
There are many more ways you can extend or update the Identity system and lots of options available, so I encourage you to explore Microsoft's "Overview of ASP.NET Core authentication" at http://mng.bz/XdGv to see your options. In the next section, you'll see how to achieve another common requirement: adding users to an existing application.

14.4 Adding ASP.NET Core Identity to an existing project

In this section we're going to add users to the recipe application from chapters 12 and 13. This is a working app that you want to add user functionality to. In chapter 15 we'll extend this work to restrict control regarding who's allowed to edit recipes on the app.

By the end of this section, you'll have an application with a registration page, a login screen, and a manage account screen, like the default templates. You'll also have a persistent widget in the top right of the screen showing the login status of the current user, as shown in figure 14.14.

The Login widget shows the current signed in user's email and a link for them to log out.



Recipe	TimeToCook	
Spaghetti Bolognese	1hrs 0mins	View

Figure 14.14 The recipe app after adding authentication, showing the login widget.

As in section 14.3, I'm not going to customize any of the defaults at this point, so we won't set up external login providers, email confirmation, or 2FA. I'm only concerned with adding ASP.NET Core Identity to an existing app that's already using EF Core.

> **TIP** It's worth making sure you're comfortable with the new project templates before you go about adding Identity to an existing project. Create a test app and consider setting up an external login provider, configuring an email provider, and enabling 2FA. This will take a bit of time, but it'll be invaluable for deciphering errors when you come to adding Identity to existing apps.

To add Identity to your app, you'll need to do the following:

1. Add the ASP.NET Core Identity NuGet packages.
2. Configure `Startup` to use `AuthenticationMiddleware` and add Identity services to the DI container.
3. Update the EF Core data model with the Identity entities.
4. Update your Razor Pages and layouts to provide links to the Identity UI.

This section will tackle each of these steps in turn. At the end of section 14.4, you'll have successfully added user accounts to the recipe app.

14.4.1 Configuring the ASP.NET Core Identity services and middleware

You can add ASP.NET Core Identity with the default UI to an existing app by referencing two NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore—Provides all the core Identity services and integration with EF Core
- Microsoft.AspNetCore.Identity.UI—Provides the default UI Razor Pages

Update your project .csproj file to include these two packages:

```
<PackageReference
    Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
    Version="5.0.0" />
<PackageReference
    Include="Microsoft.AspNetCore.Identity.UI" Version="5.0.0" />
```

These packages bring in all the additional required dependencies you need to add Identity with the default UI. Be sure to run `dotnet restore` after adding them to your project.

Once you've added the Identity packages, you can update your Startup.cs file to include the Identity services, as shown next. This is similar to the default template setup you saw in listing 14.1, but make sure to reference your existing `AppDbContext`.

Listing 14.4 Adding ASP.NET Core Identity services to the recipe app

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<ApplicationUser>(options =>
            options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<AppDbContext>();

    services.AddRazorPages();
    services.AddScoped<RecipeService>();
}
```

The existing service configuration is unchanged.

Adds the Identity services to the DI container and uses a custom user type, ApplicationUser

Makes sure you use the name of your existing DbContext app

This adds all the necessary services and configures Identity to use EF Core. I've introduced a new type here, `ApplicationUser`, which we'll use to customize our user entity later. You'll see how to add this type in section 14.4.2.

Configuring `AuthenticationMiddleware` is somewhat easier: add it to the pipeline in the `Configure` method. As you can see in listing 14.5, I've added the middleware after `UseRouting()`, just before `UseAuthorization()`. As I mentioned in section 14.3.2, it's important you use this order for middleware in your application.

Listing 14.5 Adding `AuthenticationMiddleware` to the recipe app

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // other configuration not shown
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

StaticFileMiddleware will never see requests as authenticated, even after you sign in.

Adds AuthenticationMiddleware after UseRouting() and before UseAuthorization

Middleware after AuthenticationMiddleware can read the user principal from HttpContext.User.

You've configured your app to use Identity, so the next step is to update EF Core's data model. You're already using EF Core in this app, so you need to update your database schema to include the tables that Identity requires.

14.4.2 Updating the EF Core data model to support Identity

The code in listing 14.4 won't compile, as it references the `ApplicationUser` type, which doesn't yet exist. Create the `ApplicationUser` in the Data folder, using the following line:

```
public class ApplicationUser : IdentityUser { }
```

It's not strictly necessary to create a custom user type in this case (for example, the default templates use the raw `IdentityUser`), but I find it's easier to add the derived type now rather than try to retrofit it later if you need to add extra properties to your user type.

In section 14.3.3 you saw that Identity provides a `DbContext` called `IdentityDbContext`, which you can inherit from. The `IdentityDbContext` base class includes the necessary `DbSet<T>` to store your user entities using EF Core.

Updating an existing `DbContext` for Identity is simple—update your app's `DbContext` to inherit from `IdentityDbContext`, as shown in the following listing. We're using the generic version of the base Identity context in this case and providing the `ApplicationUser` type.

Listing 14.6 Updating `AppDbContext` to use `IdentityDbContext`

Updates to inherit from the Identity context, instead of directly from DbContext

```
public class AppDbContext : IdentityDbContext<ApplicationUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    { }

    public DbSet<Recipe> Recipes { get; set; }
}
```

The remainder of the class remains the same.

Effectively, by updating the base class of your context in this way, you've added a whole load of new entities to EF Core's data model. As you saw in chapter 12, whenever EF Core's data model changes, you need to create a new migration and apply those changes to the database.

At this point, your app should compile, so you can add a new migration called `AddIdentitySchema` using

```
dotnet ef migrations add AddIdentitySchema
```

The final step is to update your application's Razor Pages and layouts to reference the default identity UI. Normally, adding 30 new Razor Pages to your application would be a lot of work, but using the default Identity UI makes it a breeze.

14.4.3 Updating the Razor views to link to the Identity UI

Technically, you don't *have* to update your Razor Pages to reference the pages included in the default UI, but you probably want to add the login widget to your app's layout at a minimum. You'll also want to make sure that your Identity Razor Pages use the same base Layout.cshtml as the rest of your application.

We'll start by fixing the layout for your Identity pages. Create a file at the "magic" path Areas/Identity/Pages/_ViewStart.cshtml, and add the following contents:

```
@{ Layout = "/Pages/Shared/_Layout.cshtml"; }
```

This sets the default layout for your Identity pages to your application's default layout. Next, add a _LoginPartial.cshtml file in Pages/Shared to define the login widget, as shown in the following listing. This is pretty much identical to the template generated by the default template, but using our custom `ApplicationUser` instead of the default `IdentityUser`.

Listing 14.7 Adding a _LoginPartial.cshtml to an existing app

```
@using Microsoft.AspNetCore.Identity
@using RecipeApplication.Data;
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
  <li class="nav-item">
    <a  class="nav-link text-dark" asp-area="Identity"
    asp-page="/Account/Manage/Index" title="Manage">Hello
    User.Identity.Name!</a>
  </li>
    <li class="nav-item">
      <form class="form-inline" asp-page="/Account/Logout"
      asp-route-returnUrl="@Url.Page("/", new { area = "" })"
      asp-area="Identity" method="post" >
        <button  class="nav-link btn btn-link text-dark"
          type="submit">Logout</button>
        </form>
    </li>
}
else
{
  <li class="nav-item">
    <a class="nav-link text-dark" asp-area="Identity"
      asp-page="/Account/Register">Register</a>
  </li>
  <li class="nav-item">
    <a class="nav-link text-dark" asp-area="Identity"
      asp-page="/Account/Login">Login</a>
  </li>
}
</ul>
```

Update to your project's namespace that contains ApplicationUser

The default template uses IdentityUser. Update to use ApplicationUser instead.

This partial shows the current login status of the user and provides links to register or sign in. All that remains is to render the partial by calling

```
<partial name="_LoginPartial" />
```

in the main layout file of your app, _Layout.cshtml.

And there you have it: you've added Identity to an existing application. The default UI makes doing this relatively simple, and you can be sure you haven't introduced any security holes by building your own UI!

As I described in section 14.3.4, there are some features that the default UI doesn't provide that you need to implement yourself, such as email confirmation and 2FA QR code generation. It's also common to find that you want to update a single page here and there. In the next section I'll show how you can replace a page in the default UI, without having to rebuild the entire UI yourself.

14.5 Customizing a page in ASP.NET Core Identity's default UI

In this section you'll learn how to use "scaffolding" to replace individual pages in the default Identity UI. You'll learn to scaffold a page so that it overrides the default UI, allowing you to customize both the Razor template and the `PageModel` page handlers.

Having Identity provide the whole UI for your application is great in theory, but in practice there are a few wrinkles, as you've already seen in section 14.3.4. The default UI provides as much as it can, but there are some things you may want to tweak. For example, both the login and register pages describe how to configure external login providers for your ASP.NET Core applications, as you saw in figures 14.12 and 14.13. That's useful information for you as a developer, but not something you want to be showing to your users. Another often-cited requirement is the desire to change the look and feel of one or more pages.

Luckily, the default Identity UI is designed to be incrementally replaceable, so that you can override a single page without having to rebuild the entire UI yourself. On top of that, both Visual Studio and the .NET CLI have functions that allow you to *scaffold* any (or all) of the pages in the default UI, so that you don't have to start from scratch when you want to tweak a page.

> **DEFINITION** *Scaffolding* is the process of generating files in your project that serve as the basis for customization. The Identity scaffolder adds Razor Pages in the correct locations so they override equivalent pages with the default UI. Initially, the code in the scaffolded pages matches that in the default Identity UI, but you are free to customize it.

As an example of the changes you can easily make, we'll scaffold the registration page and remove the additional information section about external providers. The following steps describe how to scaffold the Register.cshtml page in Visual Studio. Alternatively, you can use the .NET CLI to scaffold the registration page.[3]

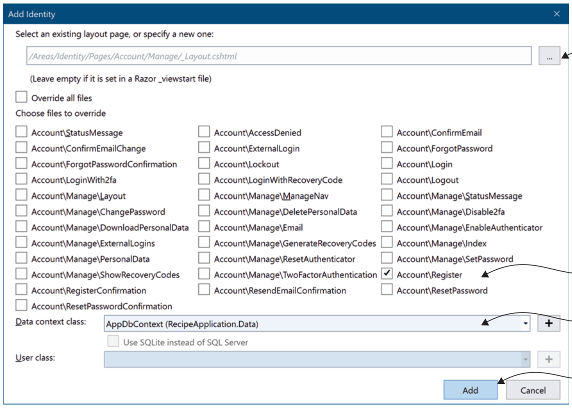
1. Add the Microsoft.VisualStudio.Web.CodeGeneration.Design and Microsoft .EntityFrameworkCore.Tools NuGet packages to your project file, if they're not already added. Visual Studio uses these packages to scaffold your application correctly, and without them you may get an error running the scaffolder.

```
<PackageReference Version="5.0.0"
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" />
<PackageReference Version="5.0.0"
    Include="Microsoft.EntityFrameworkCore.Tools" />
```

2. Ensure your project builds—if it doesn't build, the scaffolder will fail before adding your new pages.

[3] Install the necessary .NET CLI tools and packages as described in Microsoft's "Scaffold Identity in ASP.NET Core projects" documentation: http://mng.bz/yYGB. Then run `dotnet aspnet-codegenerator identity -dc RecipeApplication.Data.AppDbContext --files "Account.Register"`.

3 Right-click your project and choose Add > New Scaffolded Item.
4 In the selection dialog box, choose Identity from the category, and click Add.
5 In the Add Identity dialog box, select the Account/Register page, and select your application's `AppDbContext` as the Data context class, as shown in figure 14.15. Click Add to scaffold the page.



Add Identity ✕

Select an existing layout page, or specify a new one:

/Areas/Identity/Pages/Account/Manage/_Layout.cshtml ...

(Leave empty if it is set in a Razor _viewstart file)

□ Override all files

Choose files to override

□ Account\StatusMessage
□ Account\ConfirmEmailChange
□ Account\ForgotPasswordConfirmation
□ Account\LoginWith2fa
□ Account\Manage\Layout
□ Account\Manage\ChangePassword
□ Account\Manage\DownloadPersonalData
□ Account\Manage\ExternalLogins
□ Account\Manage\PersonalData
□ Account\Manage\ShowRecoveryCodes
□ Account\RegisterConfirmation
□ Account\ResetPasswordConfirmation

□ Account\AccessDenied
□ Account\ExternalLogin
□ Account\Lockout
□ Account\LoginWithRecoveryCode
□ Account\Manage\ManageNav
□ Account\Manage\DeletePersonalData
□ Account\Manage\Email
□ Account\Manage\GenerateRecoveryCodes
□ Account\Manage\ResetAuthenticator
□ Account\Manage\TwoFactorAuthentication
□ Account\ResendEmailConfirmation

□ Account\ConfirmEmail
□ Account\ForgotPassword
□ Account\Login
□ Account\Logout
□ Account\Manage\StatusMessage
□ Account\Manage\Disable2fa
□ Account\Manage\EnableAuthenticator
□ Account\Manage\Index
□ Account\Manage\SetPassword
☑ Account\Register
□ Account\ResetPassword

Data context class: AppDbContext (RecipeApplication.Data) +

□ Use SQLite instead of SQL Server

User class: +

Add Cancel

Leave the layout field empty.

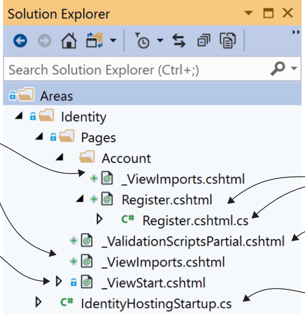
Override the Account\Register file.

Select the existing data context.

Click Add to scaffold the file.

Figure 14.15 Using Visual Studio to scaffold Identity pages. The generated Razor Pages will override the versions provided by the Default UI.

Visual Studio builds your application and then generates the Register.cshtml page for you, placing it into the Areas/Identity/Pages/Account folder. It also generates several supporting files, as shown in figure 14.16. These are mostly required to ensure your new Register.cshtml page can reference the remaining pages in the default Identity UI.



Solution Explorer

Search Solution Explorer (Ctrl+;)

Areas
Identity
Pages
Account
_ViewImports.cshtml
Register.cshtml
Register.cshtml.cs
_ValidationScriptsPartial.cshtml
_ViewImports.cshtml
_ViewStart.cshtml
IdentityHostingStartup.cs

_ViewImports files add namespace references for your custom pages.

_ViewStart sets the layout for identity UI pages to your app's layout.

The scaffolded Register.cshtml page and the Register.cshtml.cs PageModel

A duplicate of the validation scripts partial view used by the default UI

An artifact of the scaffold process that can be safely deleted.

Figure 14.16 The scaffolder generates the Register.cshtml Razor Page, along with supporting files required to integrate with the remainder of the default Identity UI.

We're interested in the Register.cshtml page, as we want to customize the UI on the Register page, but if you look inside the code-behind page, Register.cshtml.cs, you'll see how much complexity the default Identity UI is hiding from you. It's not insurmountable (we'll customize the page handler in section 14.6) but it's always good to avoid writing code if you can help it.

Now that you have the Razor template in your application, you can customize it to your heart's content. The downside is that you're now maintaining more code than you were with the default UI. You didn't have to write it, but you may still have to *update* it when a new version of ASP.NET Core is released.

I like to use a bit of a trick when it comes to overriding the default Identity UI like this. In many cases, you don't actually want to change the *page handlers* for the Razor Page, just the Razor *view*. You can achieve this by deleting the Register.cshtml.cs `PageModel` file, and pointing your newly scaffolded .cshtml file at the *original* `PageModel`, which is part of the default UI NuGet package.

The other benefit of this approach is that you can delete some of the other files that were auto-scaffolded. In total, you can make the following changes:

1. Update the `@model` directive in Register.cshtml to point to the default UI `PageModel`:

   ```
   @model Microsoft.AspNetCore.Identity.UI.V4.Pages.Account.Internal
   .RegisterModel
   ```

2. Update Areas/Identity/Pages/_ViewImports.cshtml to the following:

   ```
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
   ```

3. Delete Areas/Identity/Pages/IdentityHostingStartup.cs.
4. Delete Areas/Identity/Pages/_ValidationScriptsPartial.cshtml.
5. Delete Areas/Identity/Pages/Account/Register.cshtml.cs.
6. Delete Areas/Identity/Pages/Account/_ViewImports.cshtml.

After making all these changes, you'll have the best of both worlds—you can update the default UI Razor Pages HTML without taking on the responsibility of maintaining the default UI code-behind.

> **TIP** In the source code for the book, you can see these changes in action, where the Register view has been customized to remove the references to external identity providers.

Unfortunately, it's not always possible to use the default UI `PageModel`. Sometimes you *need* to update the page handlers, such as when you want to change the functionality of your Identity area, rather than just the look and feel. A common requirement is needing to store additional information about a user, as you'll see in the next section.

14.6 Managing users: Adding custom data to users

In this section you'll see how to customize the ClaimsPrincipal assigned to your users by adding additional claims to the AspNetUserClaims table when the user is created. You'll also see how to access these claims in your Razor Pages and templates.

Very often, the next step after adding Identity to an application is to customize it. The default templates only require an email and password to register. What if you need more details, like a friendly name for the user? Also, I've mentioned that we use claims for security, so what if you want to add a claim called IsAdmin to certain users?

You know that every user principal has a collection of claims, so, conceptually, adding any claim just requires adding it to the user's collection. There are two main times that you would want to grant a claim to a user:

- *For every user, when they first register on the app.* For example, you might want to add a Name field to the Register form and add that as a claim to the user when they register.
- *Manually, after the user has already registered.* This is common for claims used as permissions, where an existing user might want to add an IsAdmin claim to a specific user after they have registered on the app.

In this section I'll show you the first approach, automatically adding new claims to a user when they're created. The latter approach is more flexible and, ultimately, is the approach many apps will need, especially line-of-business apps. Luckily, there's nothing conceptually difficult to it; it requires a simple UI that lets you view users and add a claim through the same mechanism I'll show here.

> **TIP** Another common approach is to customize the IdentityUser entity, by adding a Name property, for example. This approach is sometimes easier to work with if you want to give users the ability to edit that property. Microsoft's "Add, download, and delete custom user data to Identity in an ASP.NET Core project" documentation describes the steps required to achieve that: http://mng.bz/aoe7.

Let's say you want to add a new Claim to a user, called FullName. A typical approach would be as follows:

1. Scaffold the Register.cshtml Razor Page, as you did in section 14.5.
2. Add a "Name" field to the InputModel in the Register.cshtml.cs PageModel.
3. Add a "Name" input field to the Register.cshtml Razor view template.
4. Create the new ApplicationUser entity as before in the OnPost() page handler by calling CreateAsync on UserManager<ApplicationUser>.
5. Add a new Claim to the user by calling UserManager.AddClaimAsync().
6. Continue the method as before, sending a confirmation email or signing the user in if email confirmation is not required.

Steps 1–3 are fairly self-explanatory and just require updating the existing templates with the new field. Steps 4–6 all take place in Register.cshtml.cs in the `OnPost()` page handler, which is summarized in the following listing. In practice, the page handler has more error checking and boilerplate; our focus here is on the additional lines that add the extra `Claim` to the `ApplicationUser`.

Listing 14.8 Adding a custom claim to a new user in the Register.cshtml.cs page

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {
            UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(
            user, Input.Password);
        if (result.Succeeded)
        {
            var claim = new Claim("FullName", Input.Name);
            await _userManager.AddClaimAsync(user, claim);

            var code = await _userManager
                .GenerateEmailConfirmationTokenAsync(user);
            await _emailSender.SendEmailAsync(
                 Input.Email, "Confirm your email", code );
            await _signInManager.SignInAsync(user);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(
                string.Empty, error.Description);
        }
    }
       return Page();
}
```

Creates an instance of the ApplicationUser entity, as usual

Validates that the provided password is valid, and creates the user in the database

Creates a claim, with a string name of "FullName" and the provided value

Adds the new claim to the ApplicationUser's collection

Sends a confirmation email to the user, if you have configured the email sender

Signs the user in by setting the HttpContext.User; the principal will include the custom claim

There was a problem creating the user. Add the errors to the ModelState, and redisplay the page.

> **TIP** Listing 14.8 shows how you can add extra claims at registration time, but you will often need to add additional data later, such as permission-related claims or other information. You will need to create additional endpoints and pages for adding this data, securing the pages as appropriate (so that users can't update their own permissions, for example).

This is all that's required to *add* the new claim, but you're not *using* it anywhere currently. What if you want to display it? Well, you've added a claim to the `ClaimsPrincipal`, which was assigned to the `HttpContext.User` property when you called `SignInAsync`. That means you can retrieve the claims anywhere you have access to the `ClaimsPrincipal`—including in your page handlers and in view templates. For example, you could display the user's `FullName` claim anywhere in a Razor template with the following statement:

```
@User.Claims.FirstOrDefault(x=>x.Type == "FullName")?.Value
```

This finds the first claim on the current user principal with a Type of "FullName" and prints the assigned value (or if the claim is not found, it prints nothing). The Identity system even includes a handy extension method that tidies up this LINQ expression (found in the System.Security.Claims namespace):

```
@User.FindFirstValue("FullName")
```

With that last tidbit, we've reached the end of this chapter on ASP.NET Core Identity. I hope you've come to appreciate the amount of effort using Identity can save you, especially when you make use of the default Identity UI package.

Adding user accounts and authentication to an app is typically the first step to customizing your app further. Once you have authentication, you can have authorization, which lets you lock down certain actions in your app, based on the current user. In the next chapter you'll learn about the ASP.NET Core authorization system and how you can use it to customize your apps; in particular, the recipe application, which is coming along nicely!

Summary

- Authentication is the process of determining who you are, and authorization is the process of determining what you're allowed to do. You need to authenticate users before you can apply authorization.
- Every request in ASP.NET Core is associated with a user, also known as a principal. By default, without authentication, this is an anonymous user. You can use the claims principal to behave differently depending on who made a request.
- The current principal for a request is exposed on HttpContext.User. You can access this value from your Razor Pages and views to find out properties of the user such as their, ID, name, or email.
- Every user has a collection of claims. These claims are single pieces of information about the user. Claims could be properties of the physical user, such as Name and Email, or they could be related to things the user has, such as HasAdminAccess or IsVipCustomer.
- Earlier versions of ASP.NET used roles instead of claims. You can still use roles if you need to, but you should use claims where possible.
- Authentication in ASP.NET Core is provided by AuthenticationMiddleware and a number of authentication services. These services are responsible for setting the current principal when a user logs in, saving it to a cookie, and loading the principal from the cookie on subsequent requests.
- The AuthenticationMiddleware is added by calling UseAuthentication() in your middleware pipeline. This must be placed after the call to UseRouting() and before UseAuthorization() and UseEndpoints().
- ASP.NET Core includes support for consuming bearer tokens for authenticating API calls and includes helper libraries for configuring IdentityServer. For

more details see Microsoft's "Authentication and authorization for SPAs" documentation: http://mng.bz/go0V.

- ASP.NET Core Identity handles low-level services needed for storing users in a database, ensuring their passwords are stored safely, and for logging users in and out. You must provide the UI for the functionality yourself and wire it up to the Identity subsystem.
- The Microsoft.AspNetCore.Identity.UI package provides a default UI for the Identity system and includes email confirmation, 2FA, and external login provider support. You need to do some additional configuration to enable these features.
- The default template for Web Application with Individual Account Authentication uses ASP.NET Core Identity to store users in the database with EF Core. It includes all the boilerplate code required to wire the UI up to the Identity system.
- You can use the `UserManager<T>` class to create new user accounts, load them from the database, and change their passwords. `SignInManager<T>` is used to sign a user in and out by assigning the principal for the request and by setting an authentication cookie. The default UI uses these classes for you, to facilitate user registration and login.
- You can update an EF Core `DbContext` to support Identity by deriving from `IdentityDbContext<TUser>`, where `TUser` is a class that derives from `IdentityUser`.
- You can add additional claims to a user using the `UserManager<TUser>.AddClaimAsync(TUser user, Claim claim)` method. These claims are added to the `HttpContext.User` object when the user logs in to your app.
- Claims consist of a type and a value. Both values are strings. You can use standard values for types exposed on the `ClaimTypes` class, such as `ClaimTypes.GivenName` and `ClaimTypes.FirstName`, or you can use a custom string, such as `"FullName"`.